Python Strings

1 Purpose

- Create strings
- Access individual characters in strings
- Access portions of strings, AKA string slicing
- Learn about string methods
- Use select string modification methods
- Use select string methods for counting, finding, and replacing parts of strings

2 Some Creative Commons Reference Sources for This Material

- *Think Python 2nd Edition*, Allen Downey, chapter 8
- *The Coder's Apprentice*, Pieter Spronck, chapter 10
- *A Practical Introduction to Python Programming*, Brian Heinold, chapter 6
- *Algorithmic Problem Solving with Python*, John Schneider, Shira Broschat, and Jess Dahmen, chapter 9

3 Creating Strings

- Objects that contain alphabetic text characters are called **strings**
- Strings can be as short as single character
- Strings do not have to contain just letters
- They can contain nearly any character (including Unicode characters like emoticons)
- Strings are defined by surrounding the text with a set of single or double quotes
- For example, `'parrot'` and `"Lumberjack"`
- Including apostrophes and/or a set of quotes within strings
  - Use an **escape sequence**
    - `\'` to include an apostrophe in a single quoted string, i.e. `'isn\'t'`
    - `\"` to include double quotes in a double quoted string, i.e. `"\"hello\""`
    - `\\` to include an actual back slash character
  - Use double quotes for strings with apostrophes in them, i.e. `"can't"`
  - Use single quotes for strings with double quotes in them, i.e. `'She said "hello."'`
  - Enclose the string between three sets of single or double quotes, i.e. `"""It's only a flesh wound," he declared"""`
  - Three sets of single or double quotes can also be used to wrap multi-line strings

String creation examples

```
>>> # string using double quotes
>>> name = "Brian Brady"
>>> name
'Brian Brady'

>>> # string using single quotes
>>> MET = 'Mechanical Engineering Technology'
>>> MET
'Mechanical Engineering Technology'

>>> # string of just numbers
>>> just_numbers = '8675309'
>>> just_numbers
'8675309'

>>> # letters, numbers, and special characters
>>> letters_nos_chars = "asdf1234!@#$"
>>> letters_nos_chars
'asdf1234!@#$'

>>> # string with an apostrophe
>>> apostrophe = "how's this"
>>> apostrophe
"how's this"

>>> # phrase with double quotes
>>> short_phrase_double = '"Get to the chopper," he yelled.'
>>> short_phrase_double
'"Get to the chopper," he yelled.'

>>> # sentence with an apostrophe and double quotes
>>> sentence = """HAL said "I'm sorry Dave, I can't do that." """
>>> sentence
'HAL said "I\'m sorry Dave, I can\'t do that." '
>>> print(sentence)
HAL said "I'm sorry Dave, I can't do that."

>>> # string spanning multiple lines
>>> three_liner = """Hello,
... is it me
... your looking for?"""
>>> three_liner
```

```
'Hello,\nis it me\nyour looking for?'
>>> print(three_liner)
Hello,
is it me
your looking for?

>>> # string with a line return between words
>>> two_words_newline = "Hello,\nWorld!"
>>> two_words_newline
'Hello,\nWorld!'
>>> print(two_words_newline)
Hello,
World!

>>> # three words with two tabs between each word
>>> three_with_tabs = "One\t\tTwo\t\tThree"
>>> print(three_with_tabs)
One        Two        Three

>>> # Unicode characters in a string
>>> # use \u for 8 bit like \u00b2 for squared
>>> # use \U for 16 bit like \U0001F600 for a smiley face
>>> # go to https://home.unicode.org to see them all
>>> squared_cubed = 'inch\u00b2 and inch\u00b3'
>>> print(squared_cubed)
inch² and inch³
```

4 String Length

- The `len()` function returns the number of characters in a string
- `len("Python")` will tell you that the string is 6 characters long

Finding a string's length

```
>>> funny_quote = "Nobody expects the Spanish Inquisition!"
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!
>>> len(funny_quote)
39
```

5 Accessing Individual String Characters

- Strings are lists of characters grouped together but treated as one object
- Access specific characters in strings based on their position in the string
- All counting in *Python* starts with zero, not one
- `"H"` in `"Hello"` is in the zeroth position (also called the zeroth index)
- Access the zeroth character using `"Hello"[0]`
- Access the first `l` using `"Hello"[2]`
- This process is referred to as **string indexing**
- The last item in `my_string` can be accessed using `my_string[len(my_string)-1]`

Slicing examples

The following examples use string indexing to access specific characters from `funny_quote`.

1. The capital **"S"** from the word **"Spanish"**
2. The exclamation mark at the end using the `len()` function
3. The letter **"x"**
4. The letter **"q"**

String indexing from the left

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> # capital S in Spanish
>>> funny_quote[19]
'S'

>>> # ! at end using len()
>>> funny_quote[len(funny_quote) - 1]
'!'

>>> # letter x
>>> funny_quote[8]
'x'

>>> # letter q
>>> funny_quote[29]
'q'
```

The following function, `string_ruler(message)`, was created to check the index positions of any letter or character in a string 100 characters long or less. Don't worry how it works at this point in time. Just use it if you need it.

A string ruler function

```
def string_ruler(message):
    new_message = spacer = '|'
    left_index_top = '|'
    left_index_bottom = '|'
    for i, letter in enumerate(message):
        new_message += letter + '|'
        spacer += ' |'
        if i > 9:
            left_index_bottom += str(i%10) + '|'
            left_index_top += str(i//10) + '|'
        else:
            left_index_bottom += str(i) + '|'
            left_index_top += ' |'
    print('string: {}'.format(new_message))
    print('        {}'.format(spacer))
    print('  left  {}'.format(left_index_top))
    print(' index: {}'.format(left_index_bottom))


>>> string_ruler("Nobody expects the...")
string: |N|o|b|o|d|y| |e|x|p|e|c|t|s| |t|h|e|.|.|.|
        | | | | | | | | | | | | | | | | | | | | |
  left  | | | | | | | | | |1|1|1|1|1|1|1|1|1|2|
 index: |0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
```

- Strings can also be indexed from the right instead of the left
- Indexing from the right uses negative numbers
- First character from the left is accessed with the index `[0]`
- The last character can be indexed directly using `[-1]`
- The negative sign says to count from the right end of the string
- Indexing from the right starts with `-1` not `-0`, because...
  - Mathematically -0 is the same as 0
  - Using -0 would likely cause confusion

String indexing diagram

```
| 0 | 1 | 2 | 3 | 4 | <-- Indexing from the left
|   |   |   |   |   |
| H | e | l | l | o | <-- String characters "Hello"
|   |   |   |   |   |
|-5 |-4 |-3 |-2 |-1 | <-- Indexing from the right
```

Examples of slicing from the right

The following examples use indexing from the **right** to access each of the requested characters from `funny_quote`.

1. The capital "**S**" from the word "**Spanish**"
2. The exclamation mark at the end
3. The letter "**x**"
4. The letter "**q**"
5. The first letter on the left using the `len()` function

String indexing from the right

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> # S using negative indexing (i.e. from the right)
>>> funny_quote[-20]
'S'

>>> # ! using negative indexing
>>> funny_quote[-1]
'!'

>>> # x using negative indexing
>>> funny_quote[-31]
'x'

>>> # q using negative indexing
>>> funny_quote[-10]
'q'

>>> # first letter using negative indexing and len()
>>> funny_quote[-len(funny_quote)]
'N'
```

6 Accessing Portions of Strings, AKA Slicing

- Part of a string (a sub-string) can be accessed in addition to single characters
- Referred to as **slicing**
- Square brackets after a string or variable with start, stop, and step values separated by colons
- A slice includes the character at the start index
- A slice ends one character before the end argument
- All three of the slice arguments are optional
    - If no step argument, it is assumed to be `1`

    - If no start argument, it is assume to be...
        - `0` for positive steps
        - Last character for negative steps
    - If no end argument, the last character included is assumed to be...
        - Last character of the string for positive steps
        - First character for negative steps
- The diagram below illustrates how slice numbering works
- Slices actually occur between characters
- Examples
    - `my_string[0:3]` and `my_string[:3]` and `my_string[0:3:1]` are the same
    - `my_string[0:]` and `my_string[0::1]` and `my_string[:]` are the same
    - `"Hello"[0:3]` would slice out `"Hel"`
    - `"Hello"[2:]` would slice out `"llo"`
    - A slice argument of `[:]` will return a copy of the entire string

String slicing diagram

```
 0   1   2   3   4   5 <-- Slice from left; my_string[1:4] => 'ell'
 |   |   |   |   |   |
 | H | e | l | l | o | <-- String characters "Hello"
 |   |   |   |   |   |
-5  -4  -3  -2  -1   | <-- Slice from right with a positive step;
 |   |   |   |   |   |       my_string[-4:-1] => 'ell'
 |   |   |   |   |   |
-6  -5  -4  -3  -2  -1 <-- Slice from right with a negative step;
                           my_string[-1:-5:-1] => 'olle'
```

Examples of slicing

Slice the following sub-strings from our string `funny_quote`. Print the string first to make it easier.

1. The first 6 characters
2. The last 12 characters
3. Every other character from the left to the right
4. The entire string written backwards (think for a second)
5. From 19 to 26

String slicing

```
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> # first 6 characters
>>> funny_quote[0:6]
```

```
'Nobody'
>>> funny_quote[:6]
'Nobody'

>>> # last 6 characters
>>> funny_quote[-6:]
'ition!'

>>> # every other from left to right
>>> funny_quote[::2]
'Nbd xet h pns nusto!'

>>> # entire string backwards
>>> funny_quote[::-1]
'!noitisiuqnI hsinapS eht stcepxe ydoboN'

>>> # from 19 to 26
>>> funny_quote[19:26]
'Spanish'
```

7 String Methods

- *Python* has a number of **methods** that can act on or with strings
- Use `x.my_method()` for a method called `my_method()` working with or on `x`
- The following code cell prints a list of string methods, functions, and operations
- The methods are the items without leading and trailing underscores
- Can also use `print(dir(str))` to list the methods

String methods list

```
>>> my_string = "Hello, World!"
>>> print(dir(my_string))
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 ↪  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 ↪  '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 ↪  '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 ↪  '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 ↪  '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 ↪  '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
 ↪  'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
 ↪  'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
 ↪  'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
 ↪  'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
 ↪  'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
 ↪  'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 ↪  'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 ↪  'translate', 'upper', 'zfill']
```

Even better, the following code will create a cleaner looking list of methods. This technique uses a list comprehension that will be covered at another time.

Concise list of string methods

```
>>> print([x for x in dir(str) if "__" not in x])
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 ↪  'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 ↪  'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 ↪  'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 ↪  'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 ↪  'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 ↪  'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 ↪  'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Not all of these methods will be investigated in this document, only some of them. Specifically, methods related to modifying strings and methods used for counting, searching, and replacing parts of strings will be reviewed.

7.1 Methods for Modifying Strings

7.1.1 Immutability of Strings

- *Python* strings are **immutable**
- Strings cannot be changed (mutated or modified) after they have been created
- To "modify" a string, a copy must be created that has the desired changes
- This is what happens with the methods that "modify" strings
  - Original string is unchanged and a new string is created with the modifications
  - "Replace" the original by assigning the new string to the original variable name

7.1.2 List of Methods for Modifying Strings

- First five of the six methods are associated with changing the case
- Their names essentially describe what they do...
  - `str.lower()`
  - `str.upper()`
  - `str.title()`
  - `str.swapcase()`
  - `str.capitalize()`
- Only one method accepts an argument
  - `str.center(x)`
  - The argument is the length of the new string in which the original will be centered
- Get help on any string method using `help(str.method_name)`, i.e. `help(str.center)`

String modification methods examples

```
>>> # The original string
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> # All lowercase letters
>>> print(funny_quote.lower())
nobody expects the spanish inquisition!

>>> print(funny_quote)   # Notice that the original did not change
Nobody expects the Spanish Inquisition!

>>> # All uppercase letters
>>> print(funny_quote.upper())
NOBODY EXPECTS THE SPANISH INQUISITION!

>>> # First letter of only the first word only is capitalized
>>> print(funny_quote.capitalize())
Nobody expects the spanish inquisition!
```

```
>>> # Swap upper and lowercase in string
>>> print(funny_quote.swapcase())
nOBODY EXPECTS THE sPANISH iNQUISITION!

>>> # Uppercase first letter in each word
>>> print(funny_quote.title())
Nobody Expects The Spanish Inquisition!

>>> # Center string, use string length + 20 characters
>>> print(funny_quote.center(len(funny_quote) + 20))
          Nobody expects the Spanish Inquisition!

>>> # Re-print the original string name
>>> # Notice that the original remained unchanged
>>> print(funny_quote)
Nobody expects the Spanish Inquisition!
```

7.2 Methods for Counting, Searching, and Replacing

- `str.count()`
  - Count the number of times that a character or sub-string appears in a larger string
  - Requires at least one argument; the sub-string to be counted (with enclosing quotes)
  - Can also include two optional arguments
    - Index position at which to start counting
    - Index immediately after the position at which to end counting
  - Examples
    - `my_string.count("a")` returns the number of times `"a"` occurs in `my_string`
    - `my_string.count("a", 2, 7)` counts starting at index 2 and ends just before index 7
    - `my_string.count("a", 4)` counts from the 4th index to the end of the string
    - It is not possible to just specify an ending index
- `str.find()`
  - Look for `string_b` within `string_a`; `string_a.find(string_b)`
  - If found, returns the index from `string_a` matching the `string_b` starting position
  - If not found, returns `-1`
  - If `string_b` occurs more than once, only the position of the first occurrence will be returned
  - Accepts optional starting and ending arguments
  - `my_string.find("the")` returns the index of the first occurrence of `"the"`
- `str.replace()`
  - Replace all occurrences of `string_b` with `string_b` within a `string_a`
  - `string_a.replace(string_b, string_c)`
  - Two required and one optional argument

        - Search string
        - New string
        - Optional argument is the number of occurrences to replace
    - `my_string.replace('I', 'you', 2)` replaces the first 2 occurrences of `'I'` with `'you'`
- The original string is not changed using any of these methods
- Use `my_string = my_string.replace('I', 'you', 2)` to modify the original string

Examples of using counting, finding, and replacing methods

The following examples are demonstrated using `funny_quote`. Notice that the original is never actually changed.

1. Find out how many times `"e"` appears in the entire string
2. Find the number of times `"is"` appears in the entire string
3. Find the number of times `"o"` appears starting with position 15
4. Find the starting location of the sub-string `"the"`
5. Find the position of `"x"` starting at index position 10
6. Replace `"Spanish"` with `"Ferris"`
7. Replace all occurrences of `"i"` with `"I"`

Counting, finding, and replacing with `funny_quote`

```
>>> # How many 'e's?
>>> funny_quote.count("e")
3

>>> # How many 'is' are there?
>>> funny_quote.count("is")
2

>>> # How many 'o's starting with 15
>>> funny_quote.count("o", 15)
1

>>> # Location of the first 'the'
>>> funny_quote.find("the")
15

>>> # Position of 'x' starting with 10
>>> funny_quote.find("x", 10)
-1

>>> # Replace 'Spanish' with 'Ferris'
>>> funny_quote.replace('Spanish', 'Ferris')
'Nobody expects the Ferris Inquisition!'
```

```
>>> # Replace every 'i' with 'I'
>>> funny_quote.replace('i', 'I')
'Nobody expects the SpanIsh InquIsItIon!'

>>> print(funny_quote)
Nobody expects the Spanish Inquisition!

>>> name = "My name is Brian"
>>> name = name.replace("Brian", "Mr. Brady")  # actually change `name`
>>> print(name)
My name is Mr. Brady
```

7.3 The "is" String Methods

- These methods ask a question of the string or portion of a string on which they act
- Partial list of the "is" string methods
  - str.islower() Are all characters lowercase?
  - str.isupper() Are all characters uppercase?
  - str.isspace() Are all characters spaces?
  - str.istitle() Does each word start with an uppercase letter with the rest lowercase?
  - str.isalpha() Are all characters alphabetic?
  - str.isalnum() Are all characters either alphabetic or numeric?
  - str.isnumeric() Are all characters numeric (0-9)?
- my_string.isalpha() asks if the entire string is composed of only alphabetic characters
  - If it does, then the method returns True
  - If not, the method returns False
- Portions of strings can be used by slicing them
  - my_string[:3].islower()
  - my_string[-4:].isnumeric()

Examples of the string "is" methods

The following code demonstrates using the "is" methods on portions of funny_quote (parts 1 to 7) and on strings containing numeric values (parts 8 and 9).

1. Is the slice from [0:6] upper case?
2. Is the slice from [0:6] title case?
3. Is the slice from [7:14] lower case?
4. Is index position 6 a space?
5. Is index position 10 a space?
6. Is the slice from [0:6] alphabetic?
7. Is the slice from [7:14] alpha-numeric?
8. Is the string "42" numeric?
9. Is the string "42.0" numeric?

The "is" methods

```
>>> # Is 0 to 6 uppercase?
>>> print(funny_quote[:6])
Nobody
>>> funny_quote[:6].isupper()
False

>>> # Is 0 to 6 title case?
>>> funny_quote[:6].istitle()
True

>>> # Is 7 to 14 lowercase?
>>> print(funny_quote[7:14])
expects
>>> funny_quote[7:14].islower()
True

>>> # Is index 6 a space?
>>> funny_quote[6].isspace()
True

>>> # Is index 10 a space?
>>> funny_quote[10].isspace()
False

>>> # Is 0 to 6 alphabetic?
>>> funny_quote[:6].isalpha()
True

>>> # Is 7 to 14 alpha-numeric?
>>> funny_quote[7:14].isalnum()
True

>>> # Is the string '42' numeric?
>>> '42'.isnumeric()
True

>>> # Is the string '42.0' numeric?
>>> '42.0'.isnumeric()
False
```